PCT/AU03/00456

AUSTRALIA

REC'D 08 MAY 2003

WIPO PCT

BEST AVAILABLE COPY

PRIORITY DOCUMENT

SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(a) OR (b)

Patent Office
Canberra

I, JONNE YABSLEY, TEAM LEADER EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. PS 1738 for a patent by CAMMS PTY LTD as filed on 16 April 2002.

COMMONWEALTH OF AUSTRALIA
AUSTRALIA
PATENT OFFICE

WITNESS my hand this
Thirtieth day of April 2003

J R Yabsley

JONNE YABSLEY
TEAM LEADER EXAMINATION
SUPPORT AND SALES

CAMMS PTY LTD

AUSTRALIA

PATENTS ACT 1990

PROVISIONAL SPECIFICATION FOR AN INVENTION ENTITLED:-

"DATA COLLECTION SYSTEM USING REMOTELY CONFIGURABLE SCRIPTING"

This invention is described in the following statement:-

TECHNICAL FIELD

This invention relates to a digital data collection system that provides the ability to configure, re-configure and append software and functionality to remote computer installations regardless of the location or Windows ™ operating system of the remote computer installation.

BACKGROUND

Issues and questions surrounding the problem of remote data collection include:

- How will specific data be retrieved?
- When is data to be retrieved?
- How is data to be sent?
- When is data to be sent?
- How can the collection system be modified remotely?
- How are the solutions to these issues going to be controlled?
- How can the collection system functions be customized?
- How can new functions (i.e. Dynamic Link Libraries) be added?
- How can old functions be replaced?
- How can new data sources be added?

Any solution to the above problems and issues should preferably not significantly impact on the operation of the remote machine from which the data is requested.

Current systems address some of the abovementioned needs and issues associated with remote data collection. A common approach is to write a single software application that is:

- Hard-coded to connect to a specific data source
- Hard-coded to connect to a temporary database
- Hard-coded to transmit data

Such an approach is typically taken because software is not available that is compatible with all the different remote systems. Different Operating Systems, different databases and different communication protocols make every solution unique

Subsequently, should any of the coded or information embedded in the hard-coded program need to be changed after installation, a rewrite and complete re-compilation of the program and an on-site installation is typically required. Changes arise for many reasons but they could be as simple as a change in a directory or as problematic as a change in the database.

The following universal needs are indicative of the shortcomings identified by current data collection systems:

- Retrieval of data from multiple remote locations
- To configure, re-configure and append installations remotely
- Non-reliance upon and independence from a particular database technology or data source type used at the remote location

It is an aim of this invention to provide a data collection system that reduces or eliminates some if not all of the shortcomings of the prior art or at least provides an alternative approach.

Brief Description of the Invention

In a broad aspect of the invention a data collection system consists of one or more objects;

a central object; and

one or more other objects; wherein

all objects have a common interface and functional elements that reference one or more common function libraries, such that a central object and at least one other object is installed at a remote data source to communicate data from said source to a remote data collection object.

In a further aspect of the invention eXtended Markup Language is the language used between objects.

In an aspect of the invention communication between objects can be synchronous or asynchronous.

In yet a further aspect of the invention said central object comprises a core system having form and central objects and at least one configuration file.

In a further aspect of the invention a base system comprises a core system as well as a transmit object, an installer object and a timer object.

In a further aspect of the invention a functional system comprises a core system as well as a store object and a connector object.

Specific embodiments of the invention will now be described in some further detail with reference to and as illustrated in the accompanying figures. These embodiments are illustrative, and not meant to be restrictive of the scope of the invention. Suggestions and descriptions of other embodiments may be included but they may not be illustrated in the accompanying figures or alternatively features of the invention may be shown in the figures but not described in the specification.

Brief Description of the Figures

Fig.1 depicts a Data Collector System Object;

Fig. 2 depicts Data Collector System Communication Environment;

Fig. 3 depicts synchronous communication between the Objects;

Fig. 4 depicts asynchronous communication between the Objects;

Fig. 5 depicts the Data Collector core system;

Fig. 6 depicts the process of command execution

Fig. 7 depicts an example of the Central Core System Object (Glue) executing and passing commands at startup;

Fig. 8 depicts a Base system;

Fig. 9 depicts the process of establishing a heartbeat;

Fig. 10 depicts the installation of a new object by the Installer object;

Fig. 11 depicts the complete functional system;

Fig. 12 depicts the retrieval of data from the remote data source and storage of this data into the local database;

Fig. 13 depicts the sending of data bundles at set time intervals; and

Fig. 14 depicts the sending of data bundles at set capacity levels.

To better understand the invention, a preferred embodiment will now be described, but it will be realized that the invention is not to be confined or restricted to the precise nature of this embodiment.

Detailed Description of an Embodiment of the Invention

The Data Collector System of the invention consists of independent, yet related, objects which together through the control imparted by one central object collects, stores and transmits data.

The central object communicates with, and co-ordinates the activity of all other objects by sending and receiving XML (eXtended Markup Language) commands. The use of XML commands is preferable. The use of multiple objects, which together perform separate functions through a common interface, enables the removal and addition of functionality. The use of XML scripting together with the ability to add and remove functionality enables remote configuration of the Data Collector System.

The Data Collector System is comprised of a collection of Data Collector System Objects that include an XML Configuration File, the Common Function Library, and a Windows Application (which hosts the Collector Objects).

Data Collector System Communication Interface

Each Data Collector System Object within the Data Collector System is comprised of two distinct parts depicted in Fig. 1.

- Common Interface
- Specialized Functionality

The Common Function Library (CFL) is accessible to all Data Collector System Objects and contains standard routines. Both the Common Interface (CI) and Specialized Functionality (SF) address the CFL yet both access a distinctly separate set of sections of the library, that is they do not share functionality within the library.

The arrangement depicted is only preferable, as it would be quite acceptable to provide the CI and SF with separate CFLs.

Common Interface

The Common Interface defines a limited and fixed set of methods/properties that each Object knows about. These are:

- SetUp
- ShutDown
- CommandXML
- ReturnCommandXML
- Error
- IsHealthy
- Ping

The properties CommandXML and ReturnCommandXML provide the functionality to pass XML commands between two Objects. Fig 1 illustrates how the Data Collector System Object is comprised of the Common Interface and Specialized Functionality parts, as well as the flow in and out of the XML commands.

An XML-Command is a single XML document, which contains a top-level command and optionally a series of sub-commands. The top-level command with sub-commands form a to-do list. The associated Object executes the top-level command first, then the first sub-command is promoted to be the top-level command and its associated Object executes it. Execution continues until all sub-commands are promoted and executed.

On receipt of an XML command, the Common Interface using the Command XML property performs the following:

1. Inspects the XML to retrieve the command and any associated parameters.
2. Calls the relevant specialized functionality that implements the command (passing parameters if required)
3. If the call returns any XML data, it adds this data to the current command.
4. Returns the (possibly updated) command to whence it came, in general the Central Core System Object (Glue), via the ReturnCommandXML property.
5. Central Core System Object (Glue) promotes the first sub-command, if one exists, and if so repeats steps 1 to 5.

Specialized Functionality

The Specialized Functionality part of the component, under instruction from the Common Interface part, performs specific operations such as connecting to a database to retrieve values or to the Internet to send information. A component may have more than one specialized operation.

COM+ is used to implement the functionality of this part of the component, and as such, it is comprised of simple procedural calls, which may or may not return data. If it does return data, does so as valid XML.

Data Collector System Communication Environment

Any object that possesses a Common Interface may be added to the Data Collector System.

The Data Collector System Communications Environment comprises one or more Data Collector System Objects and respective Common Function Libraries, plus a Configuration File and a Windows Executable. This is depicted in Fig. 2.

Intra-system Communication Protocol

Currently, Windows' COM+ technology is used to facilitate communication between Objects. Whilst it is necessary to communication between Objects, it is not necessary to use COM+. Other protocols, such as HTTP, MSMQ or SMTP, may be used.

Synchronous vs. Asynchronous Communication

Currently, communication between the Objects is synchronous, meaning that the Objects within the system may execute only one XML-Command (containing top-level & sub-commands) at any moment in time. New XML-Commands can only begin execution when all Objects become idle.

Despite this current implementation, communication between Objects may be configured so that it is asynchronous. Therefore, whilst an Object cannot execute two commands simultaneously, the system as a whole will be able to execute commands asynchronously because distinct objects would be able to execute distinct commands simultaneously. Fig. 3 –Synchronous and Fig. 4 Asynchronous show this relationship.

The Data Collector System Objects accommodate either method.

The Data Collector System preferably uses:

- An XML scripting language to co-ordinate object activities.
- A central object manages the activities of independent yet related objects.
- The Data Collector System is remotely configurable once a base system is installed as will be illustrated later in this specification.

As the arrangement of objects is completely configurable and re-configurable, the number and type of objects present within the Data Collector System at any one time can vary.

Three distinct systems that make up the Data Collector System, each comprised of different object configurations are used:

- Core system
- Base system
- Functional system

Core system

The core system is comprised of the essential objects necessary for the creation of all other objects. The core system alone creates the objects necessary to form the base system.

Base system

The base system supports, and is essential for, remote configuration such that an entire Data Collector System may be built and maintained remotely.

Functional system

The functional system addresses many of the issues concerning how the remote data is to be collected and when.

The advantages of remote data retrieval include:

- Data from multiple remote sites can be brought together

- System is completely remotely configurable
- Upgrades can be implemented "on the run"

CORE SYSTEM

The Data Collector System core system supports the following:

- Creation of new objects to form the base system

Objects

The Core System of the Data Collector System, depicted in Figure 5, is comprised of the following objects:

- Form
- (Central Core System Object herein referred to as Glue)
- Configuration File

Form

The Form object creates and holds Glue in memory. Form also contains a physical timer and sends regular time signals ("pings") to Glue. Once new objects are created, Glue in turn "pings" these objects. The ping is used for internal timings if required, however, is mostly ignored. The Timer object is the main object that utilizes the ping.

Glue

Glue binds the core system together and it in turn holds all objects of the Data Collector System in memory. It is responsible for creating all other objects of the Data Collector System and for controlling the activities of these objects through the use of XML commands. Glue executes commands addressed to it and passes commands addressed to other objects to those objects for execution. Once an object (including Glue) has executed a command, that command is passed back to Glue to indicate that execution is complete.

Glue- Commands are placed in a queue

When Glue receives a command to be executed by itself or by another object, it is placed at the end of a queue. The command at the head of the queue is called the current command and is the command currently being executed. Once execution of the current command is complete, Glue pops the next command off of the queue. This command becomes the current command.

Glue- Commands may contain sub-commands

When the current command contains sub-commands, Glue directs the entire command including the sub-commands to the object to which the current command is addressed. This object then executes the current command and once completed sends the current command together with the subcommands back to Glue.

When Glue receives an executed command back from an object, Glue checks to see if the command contains sub-commands. If the command does contain sub-commands, Glue promotes the first sub-command as the current command. The current command, together with any remaining sub-commands, is then sent to the relevant object for execution. These processes re-iterate until all sub-commands have been executed (see Fig 6). Figure 6 demonstrates the involvement of Glue in the process of command execution.

Glue- Commands may generate other commands

If the result of the execution of a command is additional commands needing to be executed, these additional commands are placed at the end of the queue within Glue, and are executed once the current and preceding commands have been executed.

Glue- Commands may return data

Commands passed back to Glue may be accompanied by data. Data returned may be referenced by associated sub-commands through use of the following tag:

```
datatype="<nameofdatatype>"
```

Once the execution of a command and its sub-commands is complete, any data returned and held is discarded and can not be used by any other commands in the queue.

Glue- CreateObject command

Glue can execute a variety of commands, however, the most important command to be executed is the CreateObject command. The CreateObject command enables Glue to create all other independent, system-related objects. Once an object is created, Glue can send commands addressed to this object for execution.

Configuration File

The configuration file stores all commands to be executed on startup and contains a list of all required initialization data. Glue reads the configuration file on startup.

XML Events that occur in the Core System

The main XML events that occur within the core system are:

1. Form creates Glue
2. Glue reads configuration file
3. Glue executes and delegates commands within configuration file, an example of which is to create objects

```
<docmd object="Glue" command="CreateObject" createobject="X" />
```

Creating an object

The CreateObject command can only be executed by Glue and is used to create all other independent and related objects. The Configuration File contains many CreateObject commands as this is read by Glue on startup and is necessary for the construction of the base system (see Figure 7). Figure 7 demonstrates the type of events, which occur on startup of the Data Collector System.

BASE SYSTEM

The Data Collector System base system supports the following:

1. Transmission of XML data from and to the client-end
2. Creation of new objects

The Data Collector System base system, depicted in Figure 8, is comprised of the core system plus the following three additional objects:

- Transmit
- Installer
- Timer

Transmit Object

The Transmit object sends and receives XML data to and from the DATA COLLECTOR SYSTEM server and as shown in Fig. 8 this is done via the Internet.

The Transmit object includes the following functions:

1. Set up the URL addresses for the sending and receiving of data.
2. Send data to the DATA COLLECTOR SYSTEM server using the Dock command.
3. Notify the DATA COLLECTOR SYSTEM server that Collector is 'alive' using the Heartbeat command.
4. Optionally encrypts and compresses data.

On receipt of a heartbeat, the DATA COLLECTOR SYSTEM server sends back a command informing the Collector to either do nothing or to perform some action. The DATA COLLECTOR SYSTEM server can only send commands if a heartbeat is received.

Installer Object

The Installer object takes a data message that contains an encoded file and recreates it. If the file is of the type DLL (Dynamic Link Library) then the installer will register the library. This functionality enables new objects to be created and existing objects to be upgraded.

Timer Object

The Timer object is responsible for storing commands to be executed at a set frequency. By synchronizing the frequency of an event with the system clock it is possible to execute commands at a specific time. Timer acts as a counter and sends stored commands at the appropriate times to Glue.

XML Events

The main XML events that occur within the base system are:

1. Setting up heartbeat URLs

```
<docmd object='Transmit' command='SetURLs'
   heartbeaturl=http://www.camms.com.au/aubot/collector/heartbeat.asp
   dockurl='http://www.camms.com.au/aubot/collector/dock.asp' />
```

2. Registering the heartbeat frequency

```
<docmd object="Timer" command="Register" interval="30"
     units="Seconds">
     <data type="Command">
          <docmd object="Transmit" command="HeartBeat" />
     </data>
</docmd>
```

3. Installation of new objects

```
<docmd object="Installer" command="Install" filename="object.dll">
     <data type="DLL">'encoded DLL'</data>
     <data type="SuccessCommand">
        <docmd object="Glue" command="AddSystemData">
             <docmd object="Transmit" command="Dock"
             datatype="Success"/>
             <data type="Success">
                  <message writeout="object.dll installed
                  successfully."/>
             </data>
        </docmd>
     </data>
     <data type="FailureCommand">
        <docmd object="Glue" command="AddSystemData">
             <docmd object="Transmit" command="Dock"
             datatype="Failure"/>
             <data type="Failure">
                  <message writeout="object.dll install failed."/>
             </data>
        </docmd>
     </data>
</docmd>
```

Establishing a heartbeat

A heartbeat sent from the Transmit object informs that the Data Collector System is 'alive'. The response from the central Data Collector System to a heartbeat is always to send a valid command, mostly a "do nothing" command. Figure 9 demonstrates the process of establishing a heartbeat.

Installing a new object

The installation process requires 3 main components:

1. XML message containing installation instructions

2. Encoded DLL
3. Success and failure commands

Figure 10 demonstrates the installation of a new object by the Installer object

FUNCTIONAL SYSTEM

The Data Collector System functional system as depicted in Fig. 11 supports the following:

1. Retrieval of data
2. Storing of data
3. Bundling of data
4. Transmission of data

Functional System Objects

The Data Collector functional system is comprised of the base system plus the following two objects:

- Store
- Connector

Store

Store is the object that manages the local database. It is necessary to collect information before sending it. Thus the information is temporarily stored in a database. Store includes functions to add items, bundle items, get a list of bundles, delete bundles, get a specific bundle and get the current bundle. This allows store to completely control the contents of the database. Store executes a command based on an accumulated size of data stored trigger or count trigger. This allows the user to configure how the database is to be managed.

Connector

The Connector object is used to connect to the remote data source. Connector has the capability of requesting a list of data items from the data source, registering groups of data items and reading those registered groups. The object has the capability of collecting events created by the data source. These events allow the Collector object

to register groups of alarms, thus when an alarm changes state, the Connector object is notified.

XML Events

The main XML events that occur within the functional system are:

1. Connector gets data from data source

```
<docmd object="Connector" command="GetInfo" groupid="accounts"/>
```

2. Store adds data to local database

```
<docmd object="Store" command="AddItem" datatype="Trend" />
```

3. Store bundles data

```
<docmd object="Store" command="BundleCurrentItems">
```

4. Store provides specific bundle

```
<docmd object="Store" command="GetBundle" bundleid="5"/>
```

5. Store provides list of bundles

```
<docmd object="Store" command="GetBundleList"/>
```

6. Transmit sends bundle

```
<docmd object="Transmit" command="Dock" datatype="Bundle"/>
```

7. Transmit sends list of bundles

```
<docmd object="Transmit" command="Dock" datatype="BundleList"/>
```

8. Store deletes old bundles

```
<docmd object="Store" command="DeleteOldBundles" olderthan="7"
units="Days" />
```

The events listed above occur whenever one of the following occurs:

1. When a set period of time has passed

```
<docmd object="Timer" command="Register" interval="30" units="Minutes"
synchtime="00:00">
    <data type="Command">
        <docmd object="Store" command="GetBundleList">
            <docmd object="Transmit" command="Dock"
            datatype="BundleList" />
        </docmd>
    </data>
</docmd>
```

2. When the local database contains a certain number of items or has reached a certain size

```
<docmd object="Store" command="SetTriggers" itemscount="108"
itemssizekbs="800">
    <data type="Command">
        <docmd object="Store" command="BundleCurrentItems">
            <docmd object="Glue" command="AddSystemData" />
            <docmd object="Transmit" command="Dock" datatype="Bundle">
        </docmd>
    </data>
</docmd>
```

3. When requested through Transmit

Retrieving data at set time intervals

The Timer object may be used to set the frequency at which specific groups of data are to be read and stored.

Figure 12 demonstrates the retrieval of data from the remote data source and storage of this data into the local database.

Sending data at set time intervals

The Timer object may be used to set the frequency at which specific groups of data are to be sent.

Figure 13 demonstrates the sending of data bundles at set time intervals.

Sending data at set capacity levels

The Store object may be used to set a capacity level at which specific groups of data are to be sent.

Figure 14 demonstrates the sending of data bundles at set capacity levels.

Alternative Transmission Methods

The Data Collector System utilizes the HTTP protocol for the transmission of data as it provides the following advantages:

1. The firewall remains open at all times allowing for the continual sending and receiving of information.

2. Data can be passed in either direction. This enables the transmission of the remotely collected data, sending of "I'm alive" messages, retrieval of update commands and the passing back of status messages.

3. A connection can be either direct or made via a proxy server.

4. An intermediate remote server is not required.

5. The general user imperative for the HTTP protocol (Internet) to remain running at all times ensures that down time for communication between Data Collector System components is minimal.

Despite these advantages, the Data Collector System may be modified to accommodate the following alternative protocols:

- MSMQ
- FTP
- SMTP

MSMQ

MSMQ (Microsoft Message Queuing) is an effective way of communicating on LAN and WAN installations as it can be configured for guaranteed delivery.

FTP

FTP (File Transfer Protocol) enables the two-way transmission of files.

SMTP

SMTP (Simple Mail Transfer Protocol) enables the transmission of emails.

It will be appreciated by those skilled in the art, that the invention is not restricted in its use to the particular application described. Neither is the present invention restricted in its preferred embodiment with regard to the particular elements and/or features described or depicted herein. It will be appreciated that various modifications

can be made without departing from the principles of the invention. Therefore, the invention should be understood to include all such modifications within its scope.

Dated this 16th day of April, 2002.

CAMMS PTY LTD
By their Patent Attorneys
MADDERNS

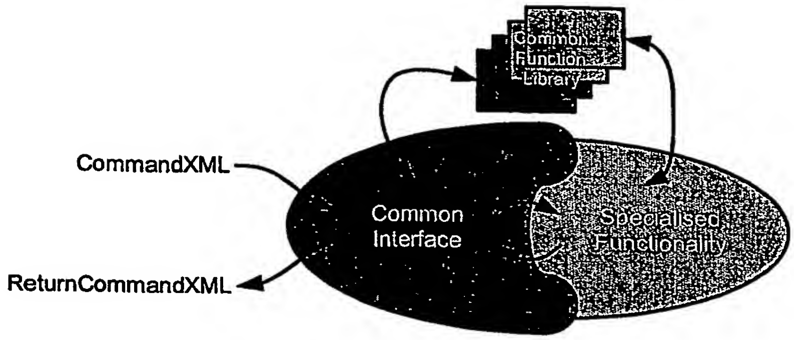


Figure 1 – Data Collector System Object

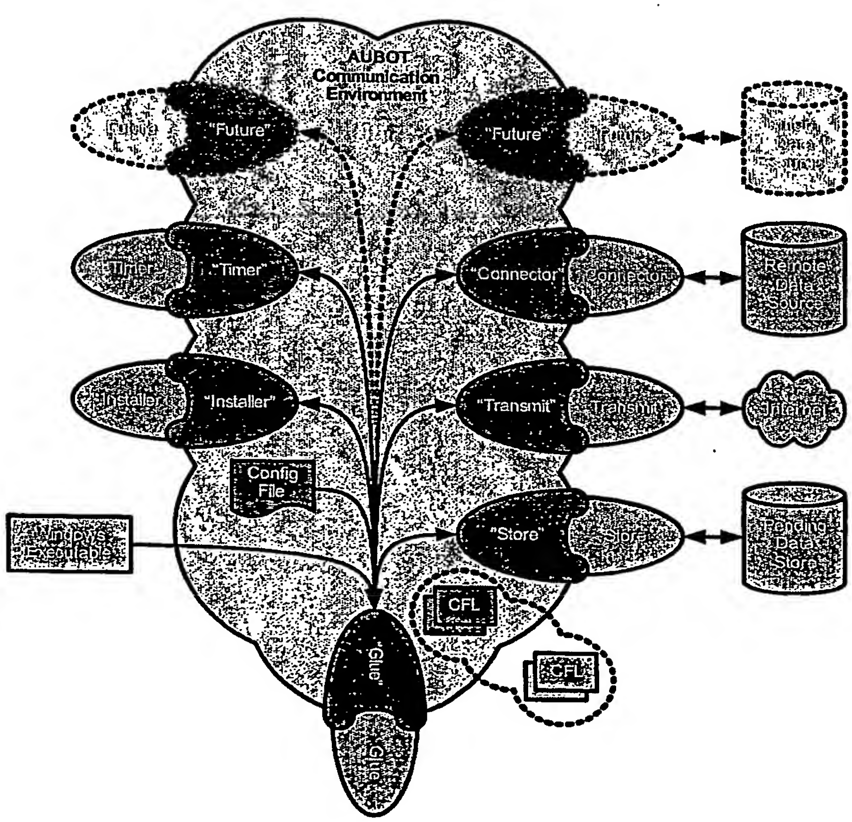


Figure 2– Data Collector System Communication Environment

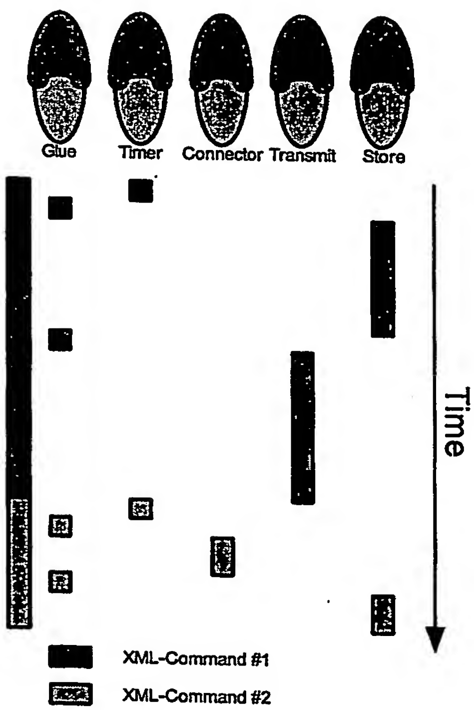


Figure 3 - Synchronous

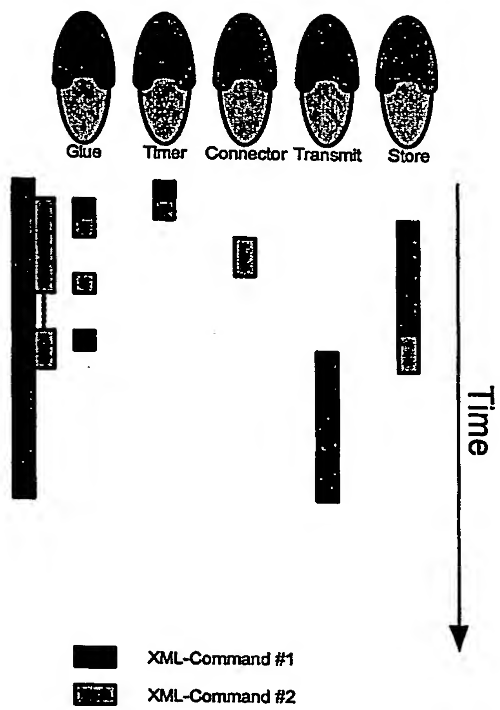


Figure 4 - Asynchronous

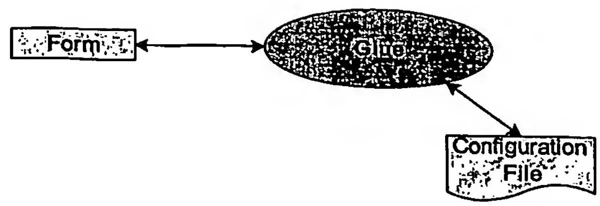


Figure 5 Core system

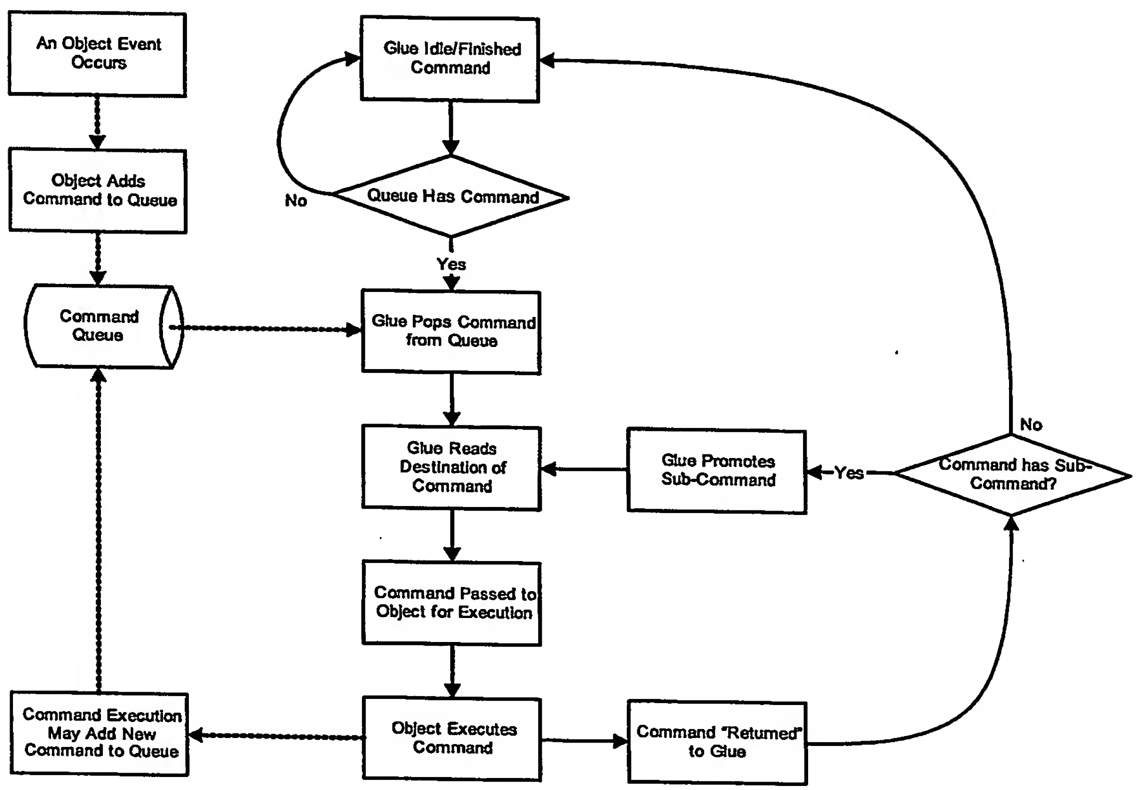


Figure 6 Process of command execution

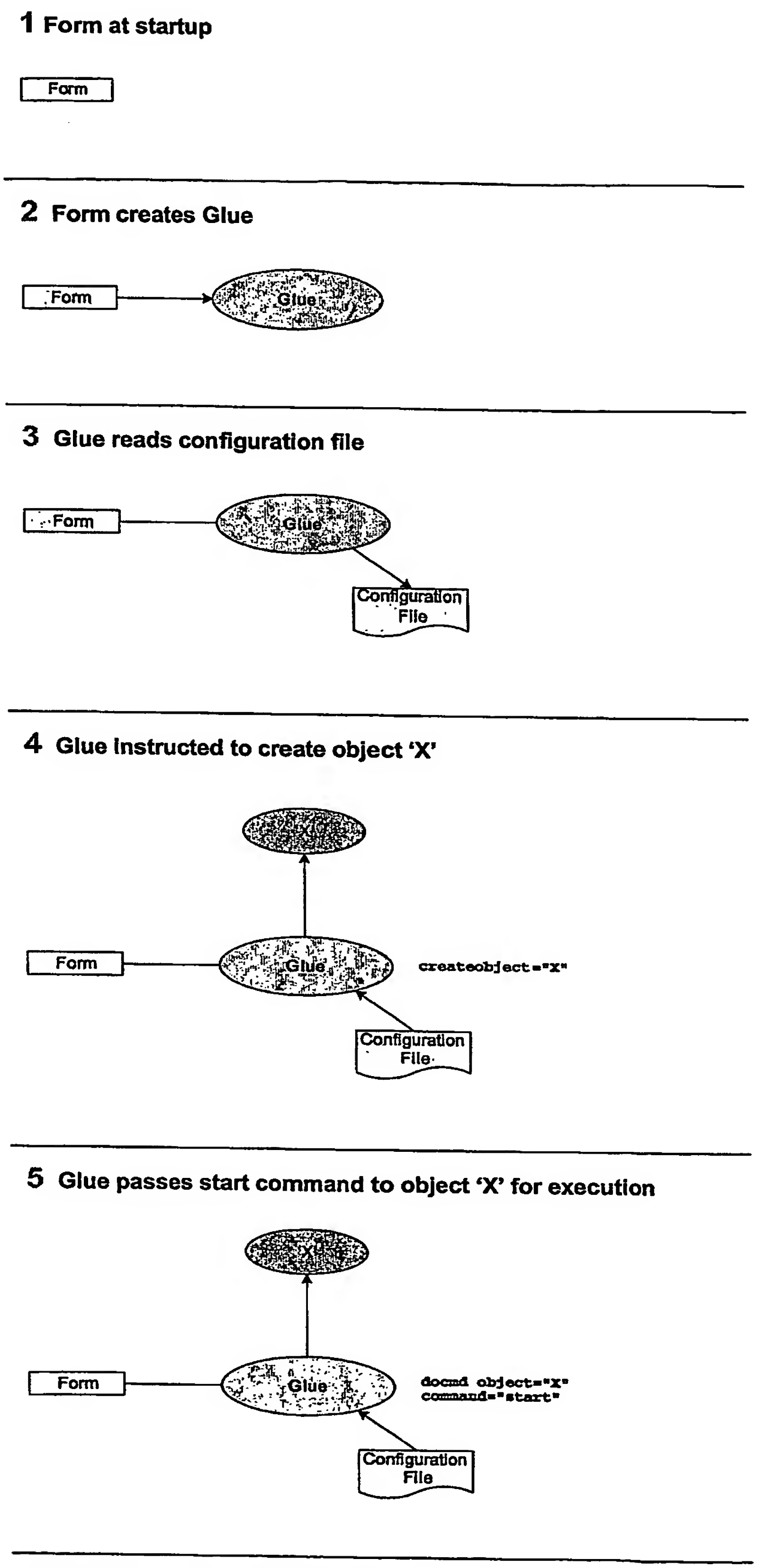


Figure 7 Example of Glue executing and passing commands at startup

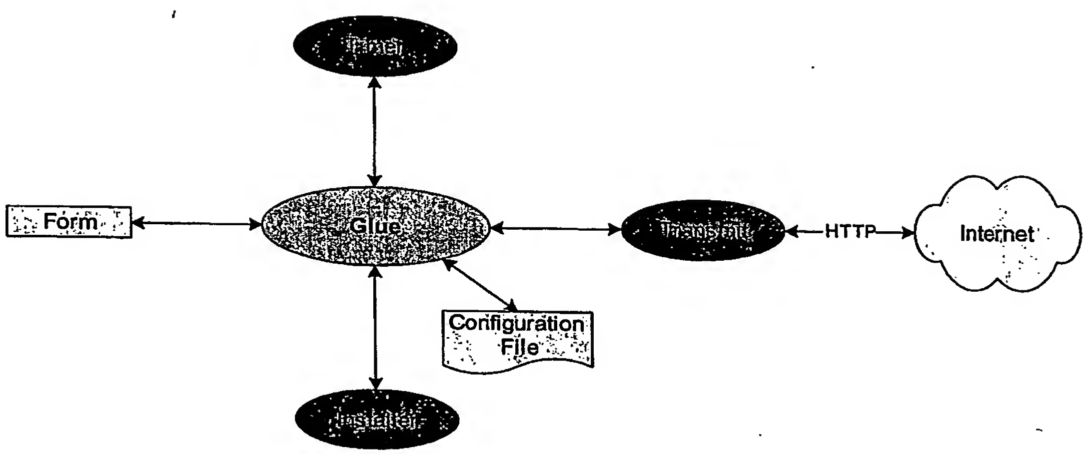
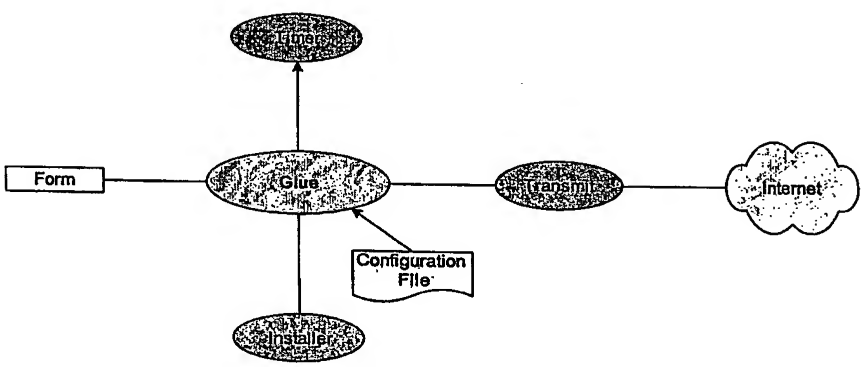


Figure 8 Base system

Figure 9 establishing a heartbeat

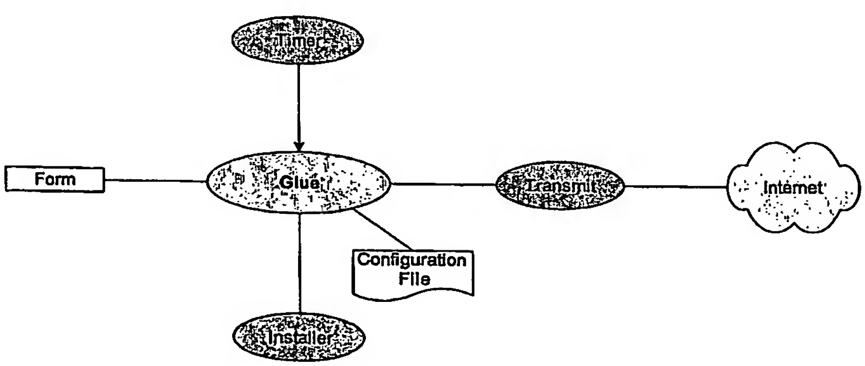
1 Timer receives instructions from Glue to send (hearbeat) command every 30 seconds

```
<docmd object="Timer" command="Register" interval="30"
   units="Seconds">
   <data type="Command">
       <docmd object="Transmit" command="HeartBeat" />
   </data>
</docmd>
```



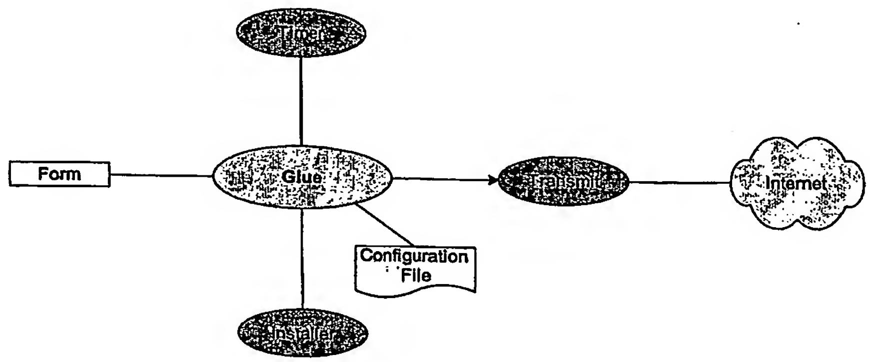
30 seconds passes

2 Timer informs Glue that there is a command to execute. Glue adds command to queue to be eventually popped off as the current command

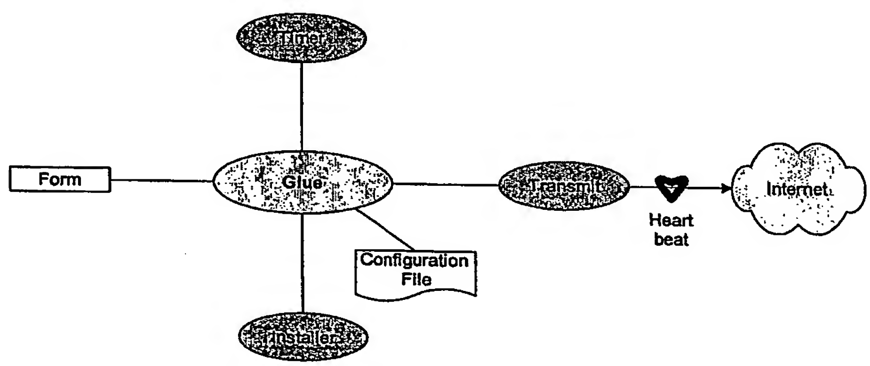


3 Glue passes command to Transmit

<docmd object="Transmit" command="HeartBeat" />

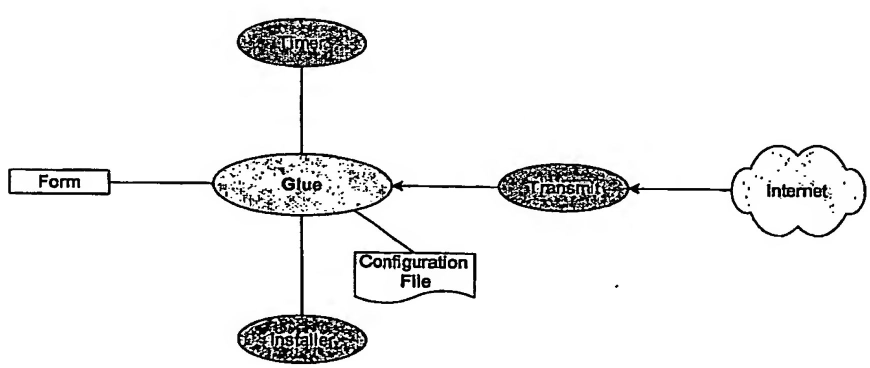


4 Transmit sends heartbeat



5a Heartbeat returns with "Nothing"

<docmd object="Glue" command="Nothing" />



5b Heartbeat returns with commands to execute

```
<docmd object="Installer" command="Install" filename="object.dll">
    <data type="DLL">'encoded DLL'</data>
    <data type="SuccessCommand">
        <docmd object="Glue" command="AddSystemData">
            <docmd object="Transmit" command="Dock" datatype="Success"/>
            <data type="Success">
                <message writeout="object.dll installed successfully."/>
            </data>
        </docmd>
    </data>
    <data type="FailureCommand">
        <docmd object="Glue" command="AddSystemData">
            <docmd object="Transmit" command="Dock" datatype="Failure"/>
            <data type="Failure">
                <message writeout="object.dll install failed."/>
            </data>
        </docmd>
    </data>
</docmd>
```

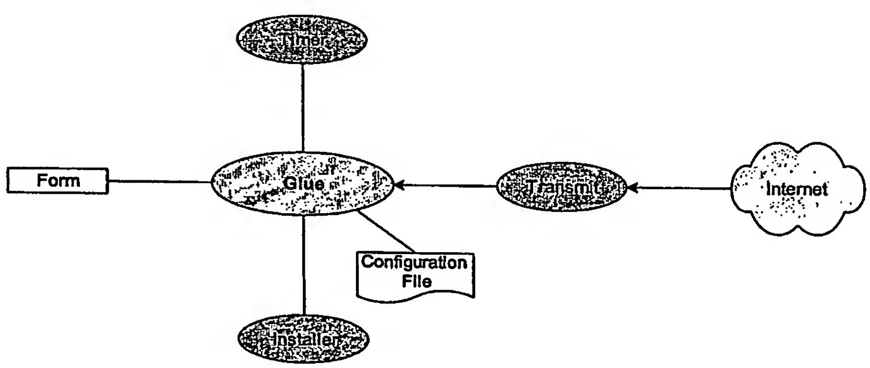
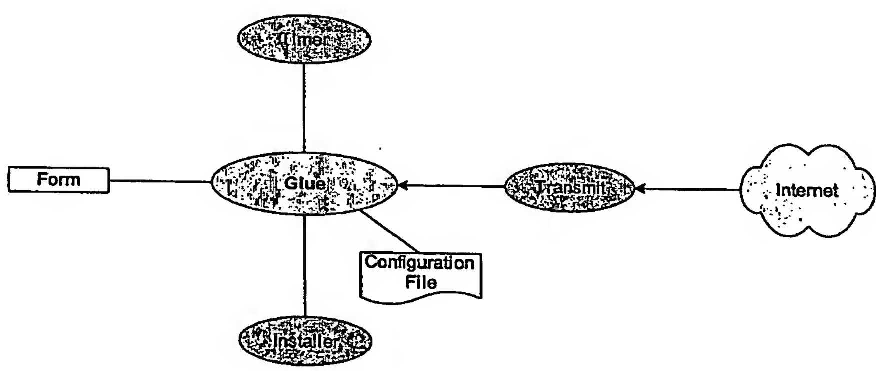


Figure 9 establishing a heartbeat

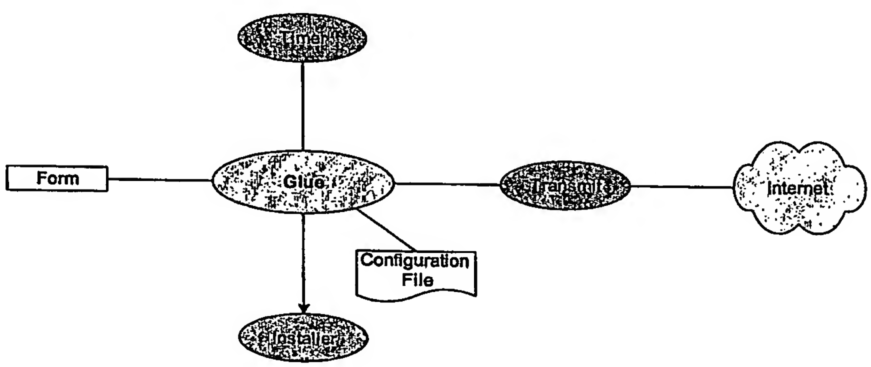
Figure 10 Installation of new object

1 Transmit receives command and passes to Glue. Glue adds command to queue to be eventually popped off as the current command



2 **Glue passes command to Installer**

```
<docmd object="Installer" command="Install" filename="object.dll">
    <data type="DLL">'encoded DLL'</data>
    <data type="SuccessCommand">
        <docmd object="Transmit" command="Dock" datatype="Success"/>
            <data type="Success">
                <message writeout="object.dll installed successfully."/>
            </data>
    </data>
    <data type="FailureCommand">
        <docmd object="Transmit" command="Dock" datatype="Failure"/>
            <data type="Failure">
                <message writeout="object.dll install failed."/>
            </data>
    </data>
</docmd>
```



3 **Installer attempts to install new object. As install was successful, Installer passes the command contained within the "SuccessCommand" datatype to the queue. The current command is passed back to Glue**

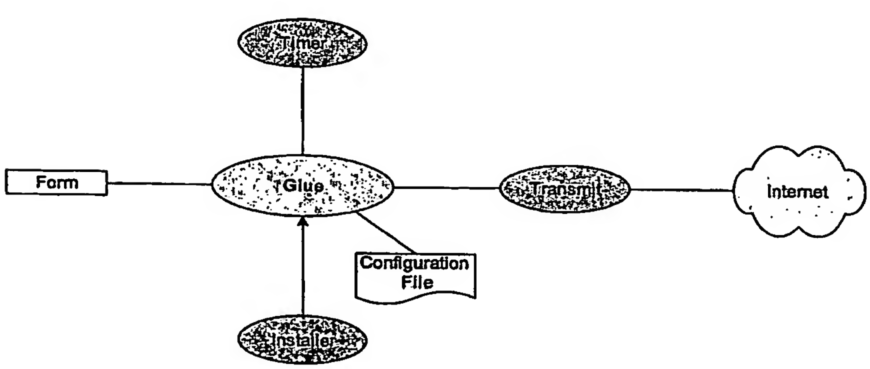
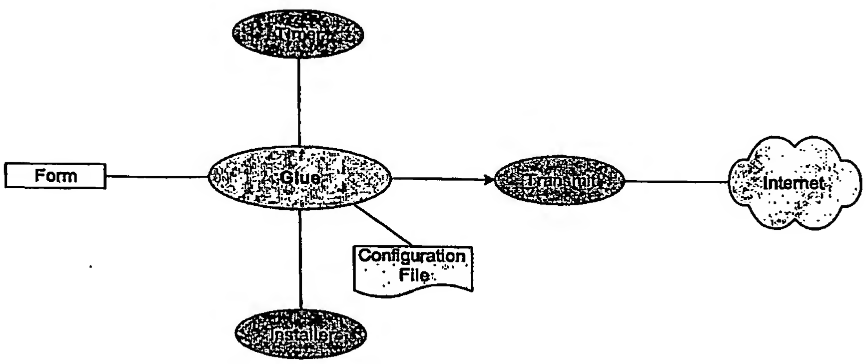


Figure 10 Installation of new object

4 Glue completes execution of the current command. Eventually the "success" command is popped off as the current command and passed to Transmit

```
<docmd object="Transmit" command="Dock" datatype="Success"/>
   <data type="Success">
       <message writeout="object.dll installed successfully."/>
   </data>
```



5 Transmit sends status and then passes the command back to Glue. As there are no more sub-commands, Glue ceases execution and pops the next command off of the queue

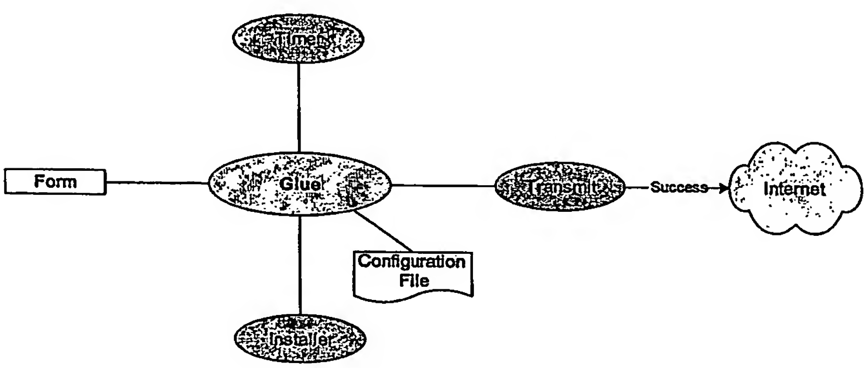


Figure 10 Installation of new object

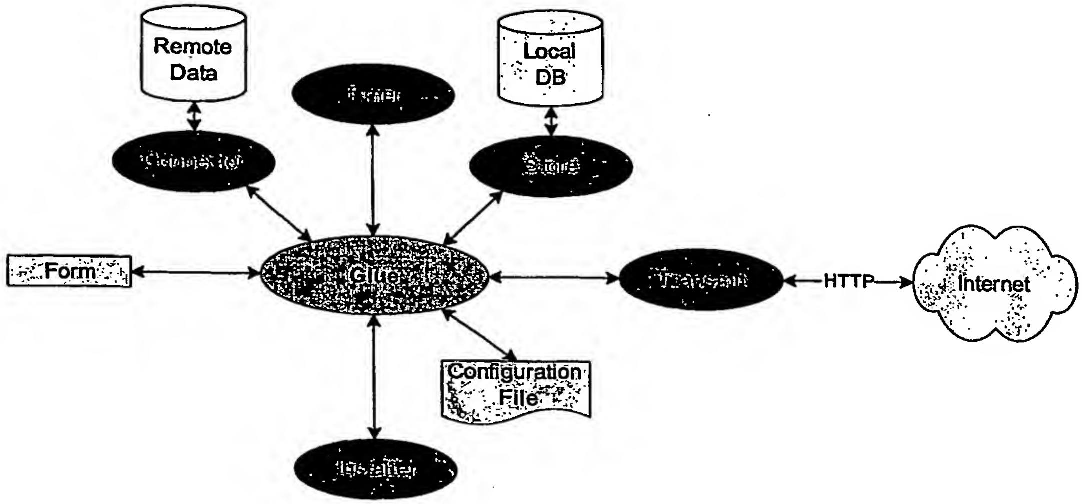
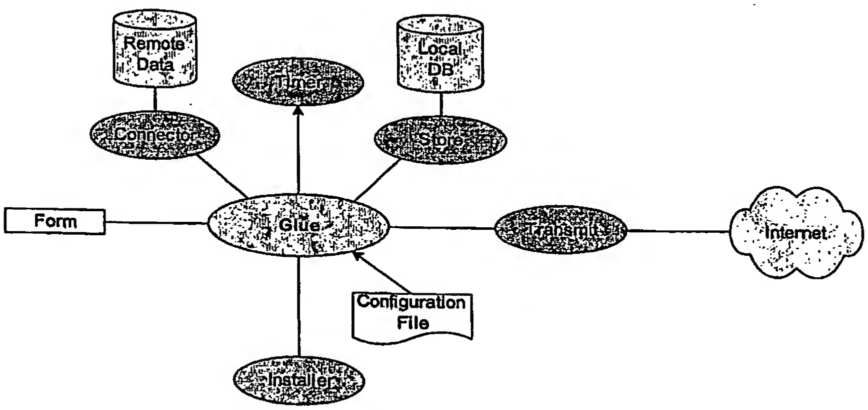


Figure 11 Functional system

Figure 12 Retrieving data at set time intervals

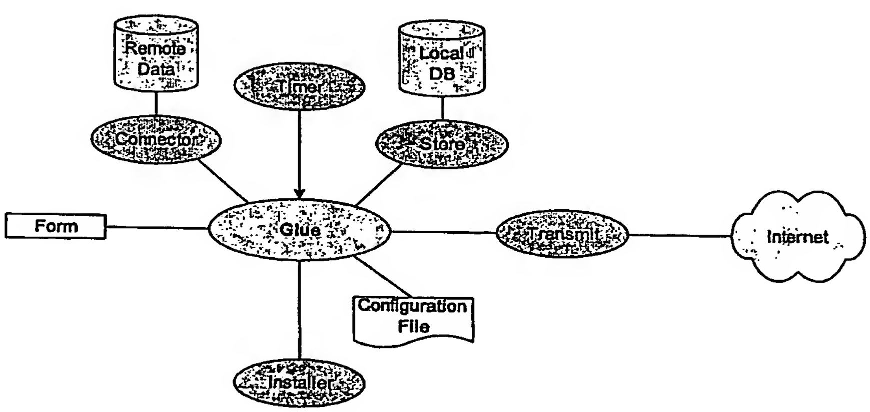
1 Timer receives instructions from Glue to send (Get-Add) command every minute

```
<docmd object="Timer" command="Register" interval="1" units="Minutes"
  synchtime="00:00">
  <data type="Command">
      <docmd object="Connector" command="GetInfo">
          <docmd object="Store" command="AddItem" datatype="Info" />
      </docmd>
  </data>
</docmd>
```



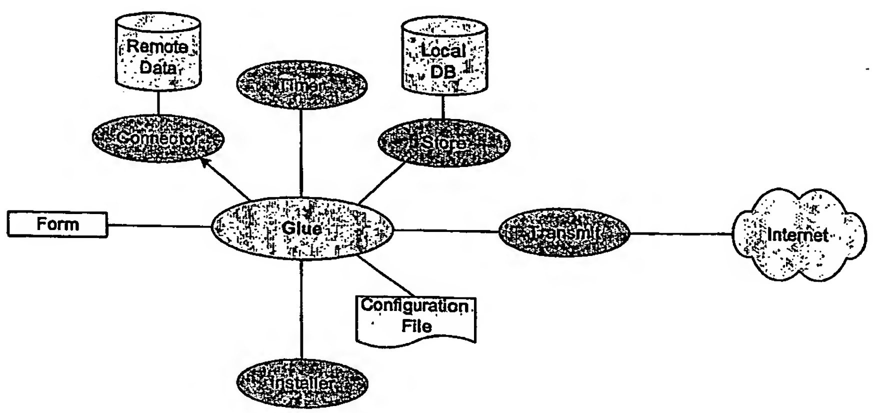
1 minute passes

2 Timer informs Glue that there is a command to execute. Glue adds command to queue to be eventually popped off as the current command



3 Glue passes "GetInfo" command to Connector

```
<docmd object="Connector" command="GetInfo">
   <docmd object="Store" command="AddItem" datatype="Info" />
</docmd>
```



4 Connector retrieves data from remote data source and passes to Glue with current command. Glue checks if command contains sub-commands, and then promotes the first sub-command as the current command

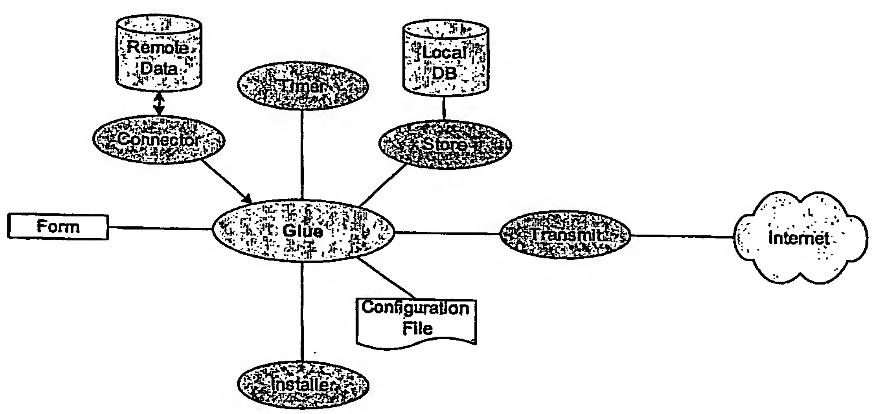
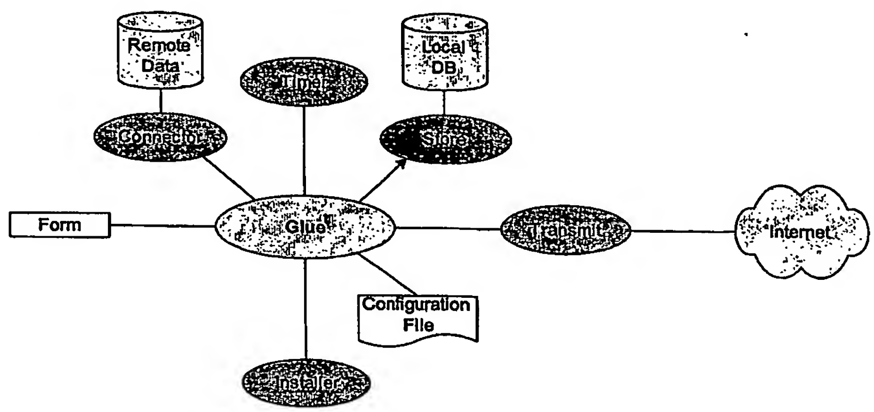


Figure 12 Retrieving data at set time intervals

5 **Glue passes "AddItem" command to Store**

```
<docmd object="Store" command="AddItem" datatype="Info" />
```



6 **Store adds data to local database and passes message back to Glue. As there are no more sub-commands, Glue ceases execution and pops the next command off of the queue**

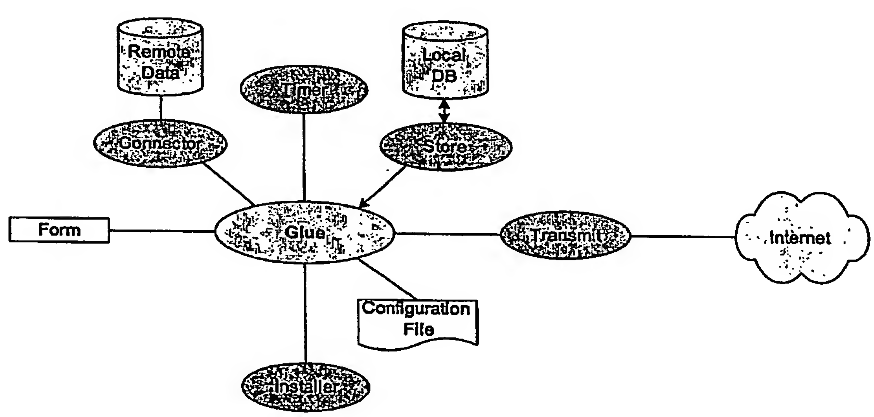
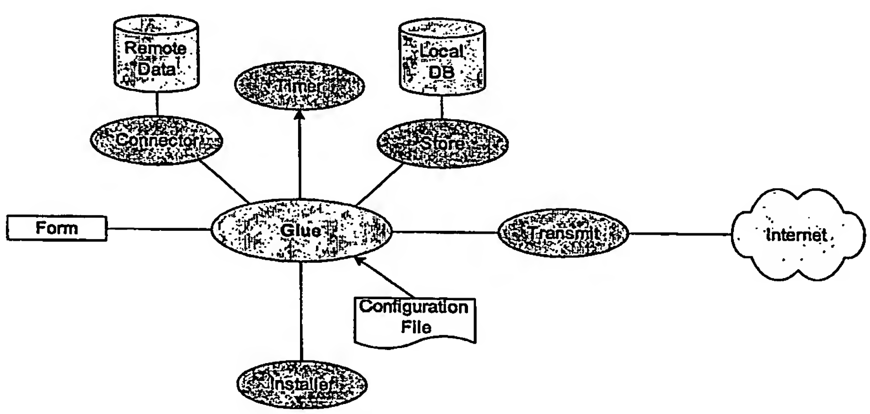


Figure 12 Retrieving data at set time intervals

Figure 13 Sending data at set time intervals

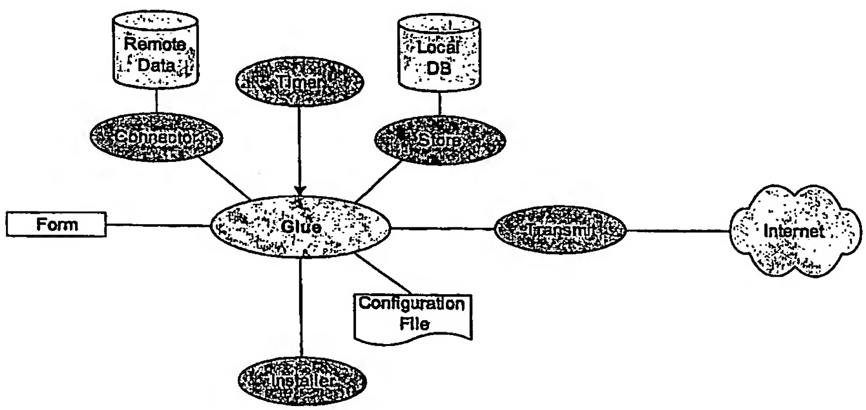
1 Timer receives instructions from Glue to send a (Bundle-Dock) command every four hours

```
<docmd object="Timer" command="Register" interval="4" units="Hours"
synchtime="00:00">
    <data type="Command">
        <docmd object="Store" command="BundleCurrentItems">
            <docmd object="Transmit" command="Dock" datatype="Bundle"/>
        </docmd>
    </data>
</docmd>
```



4 hours pass

2 Timer informs Glue that there is a command to execute. Glue adds command to queue to be eventually popped off as the current command



3 Glue passes "BundleCurrentItems" command to Store

```
<docmd object="Store" command="BundleCurrentItems">
    <docmd object="Transmit" command="Dock" datatype="Bundle" />
</docmd>
```

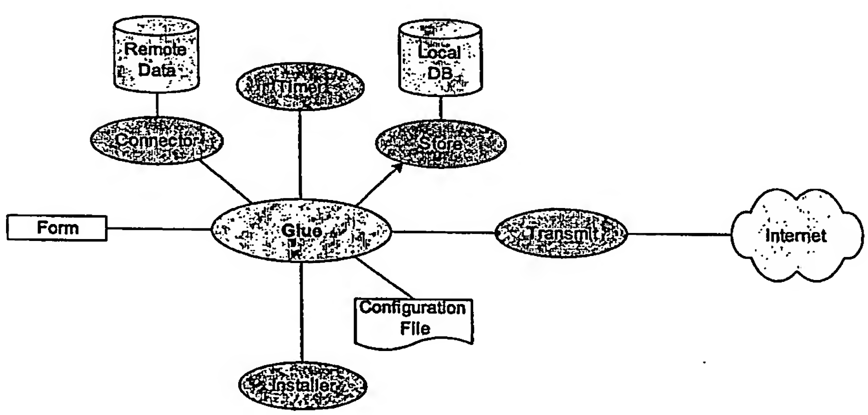
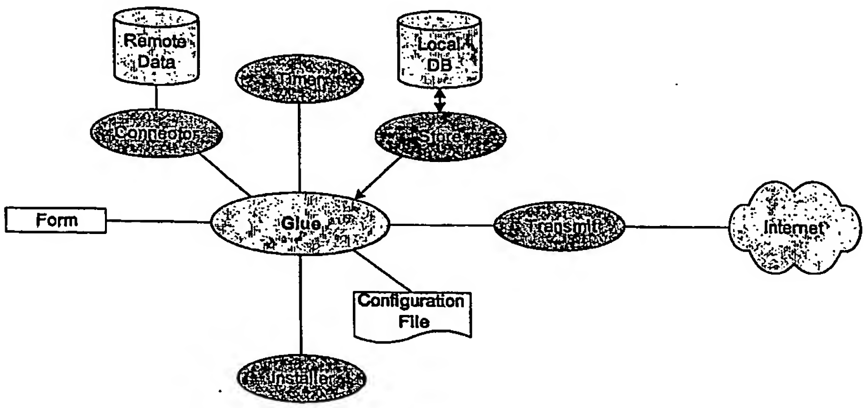


Figure 13 Sending data at set time intervals

4 **Store bundles current items in the local database and passes bundle and current command back to Glue. Glue checks if command contains sub-commands, and then promotes the first sub-command as the current command**



5 **Glue passes "Dock" command to Transmit**

```
<docmd object="Transmit" command="Dock" datatype="Bundle" />
```

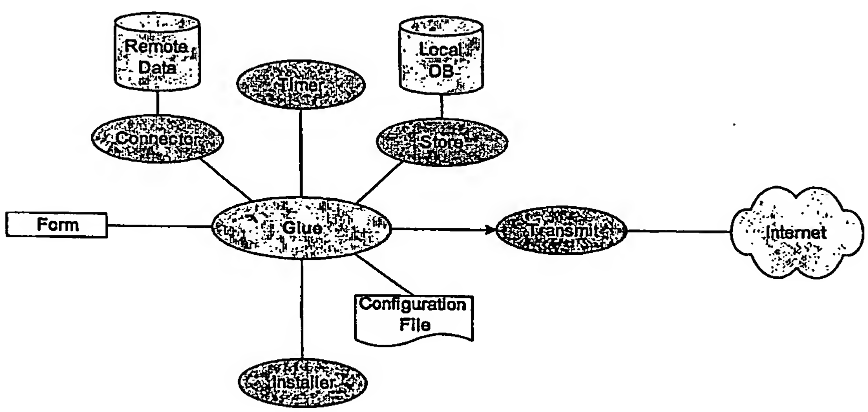
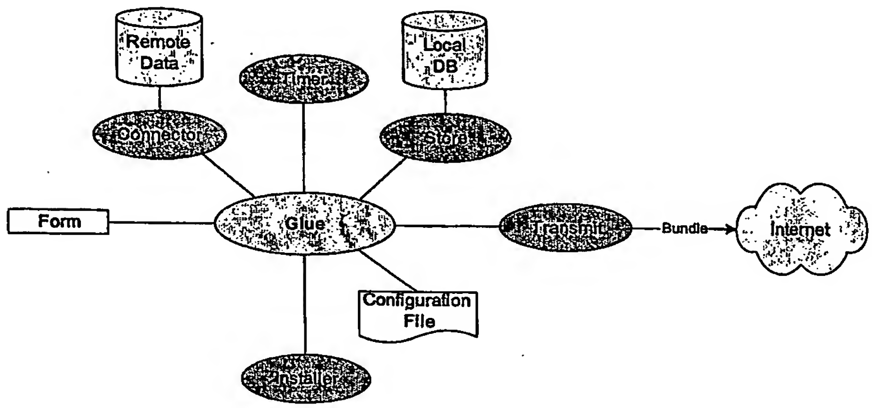


Figure 13 Sending data at set time intervals

6 Transmit sends bundle



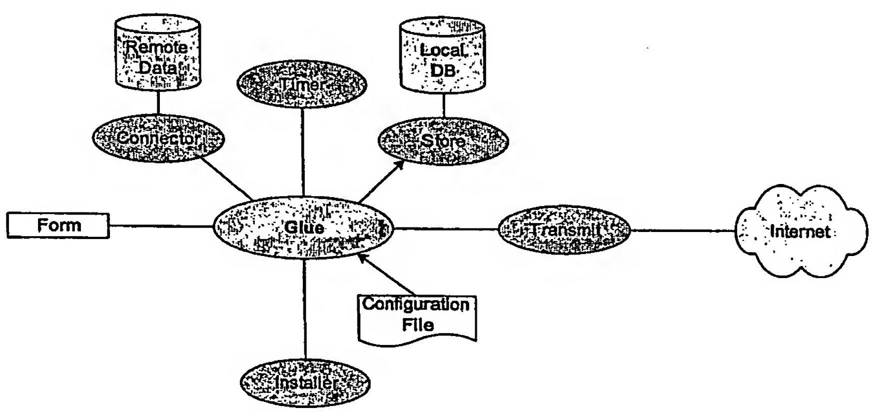
7 Transmit returns bundle and commands to Glue. As there are no more sub-commands, Glue ceases execution and pops the next command off of the queue

Figure 13 Sending data at set time intervals

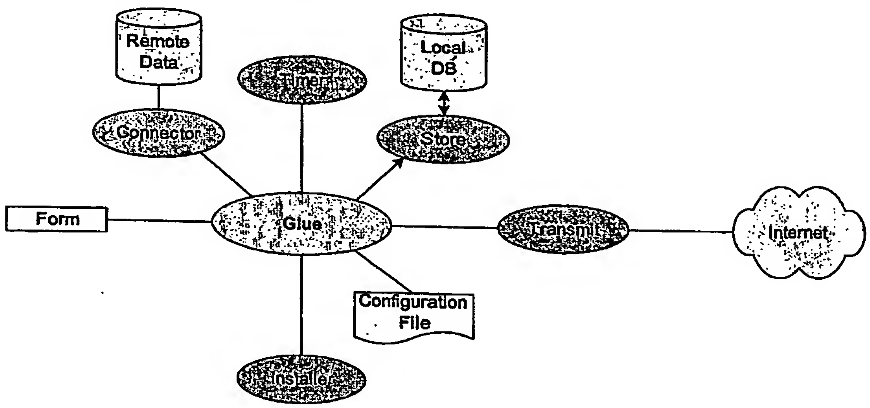
Figure 14 the sending of data bundles at set capacity levels.

1 Store receives instructions from Glue to send a (Bundle-Dock) command when local data base contains 200 data items

```
<docmd object="Store" command="SetTriggers" itemscount="200" itemssizekbs="800">
   <data type="Command">
         <docmd object="Store" command="BundleCurrentItems">
              <docmd object="Transmit" command="Dock" datatype="Bundle">
         </docmd>
   </data>
</docmd>
```



2 200th data item is added to the local database



3 Store processes own commands first (ie. bundles) then informs Glue that there is a command to execute. Glue adds command to queue to be eventually popped off as the current command

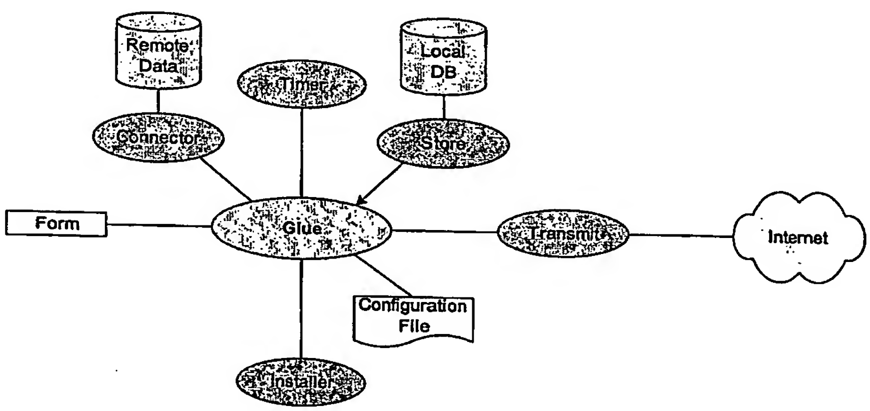
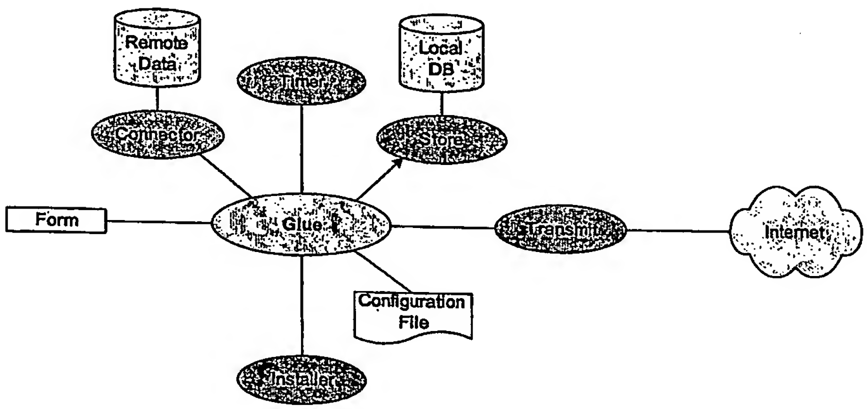


Figure 14 the sending of data bundles at set capacity levels.

4 **Glue passes "BundleCurrentItems" command to Store**

```
<docmd object="Store" command="BundleCurrentItems">
   <docmd object="Transmit" command="Dock" datatype="Bundle">
</docmd>
```



5 **Store bundles items in local database and passes back to Glue with command. Glue checks if command contains sub-commands, and then promotes the first sub-command as the current command**

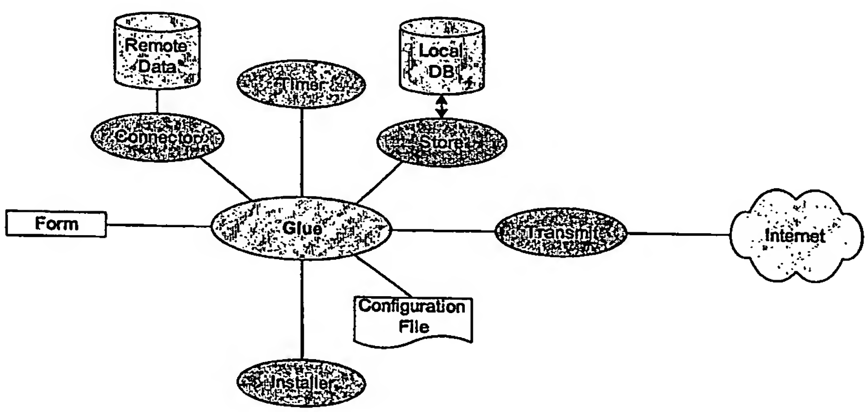
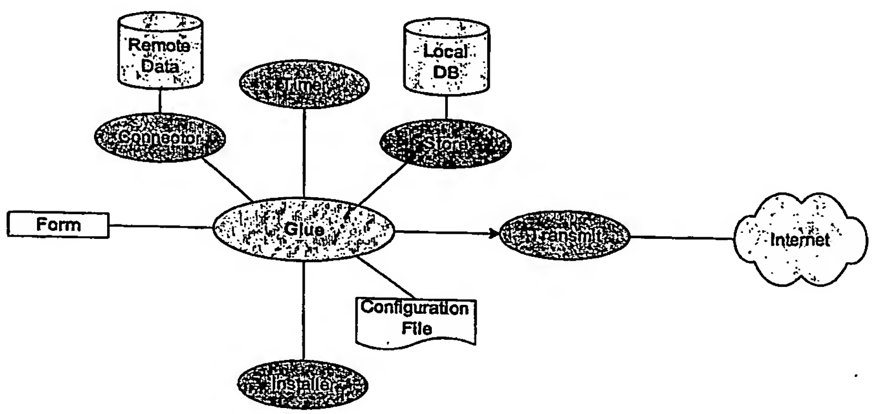


Figure 14 the sending of data bundles at set capacity levels.

6 Glue passes "Dock" command to Transmit

```
<docmd object="Transmit" command="Dock" datatype="Bundle">
```



7 Transmit sends bundle and passes command back to Glue. As there are no more sub-commands, Glue ceases execution and pops the next command off of the queue

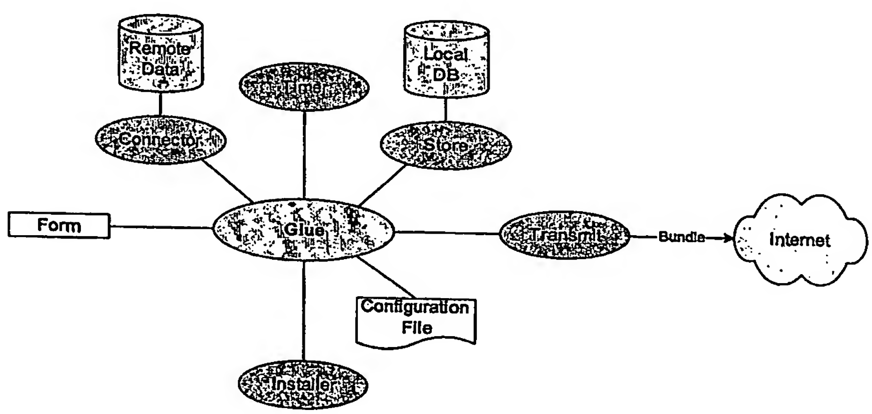


Figure 14 the sending of data bundles at set capacity levels.